VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

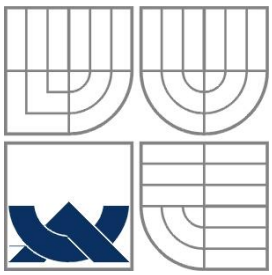
TESTOVÁNÍ VÝKONNOSTI JAVA KOLEKCÍ NA VÍCEJÁDROVÝCH SYSTÉMECH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

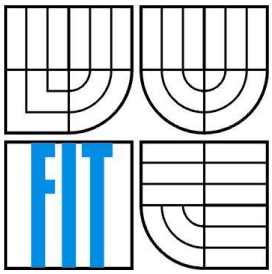
AUTOR PRÁCE
AUTHOR

MARTIN HUSAR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

TESTOVÁNÍ VÝKONNOSTI JAVA KOLEKCÍ NA VÍCEJÁDROVÝCH SYSTÉMECH

JAVA COLLECTIONS PERFORMANCE TESTING ON MULTICORE SYSTEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN HUSAR

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ZDENĚK LETKO, Ph.D.

BRNO 2013

Abstrakt

Tato práce se zabývá testováním výkonnosti Java kolekcí na vícejádrových systémech. Cílem práce bylo nastudovat kolekce z rámce Java Collection Framework a také některé další kolekce z balíku java.util.concurrent a projektu Javolution. Pro tyto kolekce bylo za úkol navrhnout vhodné zátěžové testy, na základě kterých bude možné porovnat výkonnost jednotlivých kolekcí. Základní řešení obnáší implementaci navržených testů v jazyce Java a jejich závěrečné vyhodnocení.

Abstract

This thesis deals with Java collections performance testing on multicore systems. The aim of the work is studying of core collections contained in Java Collection Framework and studying also some other collections from package java.util.concurrent and third-party project known as Javolution. The task was to suggest appropirate performance test for these collections and create based on its results a comparison. Afterwards the basic solution is an implementation of suggested tests in Java programming language and evaluation of achieved results.

Klíčová slova

Java kolekce, Java Collection Framework, ArrayList, LinkedList, HashMap, TreeMap, Javolution, FastList, FastMap, CopyOnWriteArrayList, ConcurrentHashMap, vícevláknový přístup, testování výkonnosti, měření výkonnosti, zátěžové testy

Keywords

Java Collections, Java Collection Framework, ArrayList, LinkedList, HashMap, TreeMap, Javolution, FastList, FastMap, CopyOnWriteArrayList, ConcurrentHashMap, concurrent access, performance testing, measuring performance, load tests

Citace

Husar Martin: Testování výkonnosti Java kolekcí na vícejádrových systémech, bakalářská práce, Brno, FIT VUT v Brně, 2013

Testování výkonnosti Java kolekcí na vícejádrových systémech

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Zdeňka Letka. Další informace mi poskytl Pavel Tišnovský jakožto konzultant z firmy Red Hat Czech, s.r.o.Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Husar
31. července 2013

Poděkování

Rád bych poděkoval své rodině, která mě vždy podporovala po celou dobu mých studií a držela mi palce, abych je už konečně dokončil. Mé díky patří taky vedoucímu této práce za jeho čas a cenné rady, které mi s ochotou poskytl.

Obsah

1 Úvod

Ve výpočetní technice slouží datové struktury pro ukládání dat v počítači. Počítačové programy používají datové struktury pro uchovávání těchto dat ve své operační paměti a k jejich zpracování. Pečlivá volba vhodných datových struktur vede k efektivnímu zpracování dat a účinnějším algoritmům programu. Důležitými operacemi nad datovými strukturami jsou vkládání, vyhledávání a získávání informací.

Téma této bakalařské práce je testování výkonnosti Java kolekcí na vícejádrových systémech. Testování je neodmyslitelnou součastí vývoje softwaru. Vykonává se, protože programy jsou stejně nedokonalé jako lidé, kteří je píší. Narozdíl od testování zaměřené na hledání chyb v programu, mají testy výkonnostní odlišné cíle, kdy můžeme například měřit čas odezvy aplikace s grafickým uživatelským rozhraním.

Cílem práce je popsat a porovnat interní způsob reprezentace datových kolekcí v jazyku Java, navrhnout pro ně vhodné zátěžové testy pro měření výkonnosti základních operací nad těmito kolekcemi při vícevláknovém přístupu. Úkolem bylo tyto testy naimplementovat.

Práce je rozdělená celkem do 8 kapitol. V následující kapitole *Úvod do vícevláknového programování* budou vysvětleny základní pojmy jako je vlákno a také problémy související s vícevláknovým programováním a synchronizace vláken. Kapitola 3 - *Testování vícevláknových programů* se věnuje testování programů. Kapitola 4 - *Java Collection Framework* se zaměřuje na popis jednotlivých datových kolekcí, plus popisuje také některé vybrané kolekce z balíku Javolution. Kapitola 5 - *Návrh vhodných zátěžových testů pro měření výkonnosti datových kolekcí* obsahuje návrh testů pro kolekce implementující rozhraní `Map` a `List`. V 6. kapitole - *Implementace zátěžových testů datových kolekcí v jazyce Java* budou představeny použité pragramovací prostředky, využité knihovny, způsob implementace a popis nejdůležitějších tříd. Vyhodnocení výsledků testování je popsáno v kapitole 7 - *Zhodnocení výsledků dosažených při testování výkonnosti Java kolekcí*. Poslední kapitola obsahuje *Závěr*.

2 Úvod do vícevláknového programování

Pro programátora je složité napsat korektně fungující program; napsat korektně fungující, vícevláknovou aplikaci je mnohem složitější. Tento fakt si můžeme jednoduše vysvětlit tím, že ve vícevláknových aplikacích je daleko vetší riziko, že něco bude fungovat špatně, než v programech vykonávaných sekvenčně. Proč bychom se tedy měli zabývat složitým a na chyby velmi náchylným paralelním zpracováním? Odpovědí na tuto otázku jsou bezpochyby vlákna, která představují nutnou součást dnešních programovacích jazyků jako je například i Java. Vlákna mohou účinně přispět ke zjednodušení vývoje složitých, komplexních systémů a při programování vedou k lépe srozumitelnému a čitelnému kódu. Použití vláken je kromě toho nejjednodušší způsob, jak využít výpočetní sílu na vícejádrových systémech. Problematika paralelního zpracování pomocí vláken na vícejádrových systémech je v současné době, kdy je kladen důraz zejména na efektivní využití výpočetní síly při stále se zvyšujícím počtu procesorů, velmi aktuální. [2]

2.1 Vlákna

Java umožňuje paralelní běh běh dvou či více částí programu neboli tzv. *multithreading*. Každá paralelně běžící část programu se v Javě nazývá vlákno (angl. *thread*). Důležité je, že jednotlivá vlákna lze naprogramovat téměř nezávisle a pouze v případě, že sdílí společná data nebo používají společné prostředky (zařízení), je třeba zajistit jejich řádnou synchronizaci.

Vlákno se může nacházet ve čtyřech stavech popsaných v [3]:

- *Běžící* - Vlákno se provádí.
- *Pozastavené* - Provádění vlákna je pozastaveno a může se obnovit od místa, ve kterém bylo zastaveno.
- *Blokované* - Nelze přistoupit ke sdílenému zdroji, protože jej používá jiné vlákno.
- *Ukončené* - Provádění vlákna bylo zastaveno a nelze jej obnovit.

Všechna vlákna si nejsou rovna. Některá jsou důležitější než jiná a je jim dána vyšší priorita pro přístup ke zdrojům, jakým je například prováděcí čas procesoru. Každému vláknu se přidělí priorita, podle které se určuje, kdy se má přepnout provádění z jednoho vlákna na druhé. Tomu říkáme *přpínání kontextu*.

2.1.1 Výhody využití vláken

Při jejich správném použití, mohou vlákna zmírnit náklady na vývoj a údržbu a vést ke zlepšení výkonu složitých aplikací. Mohou také pomoci převést jinak komplikovaný kód na jednodušší a lépe čitelný kód. Zlepšují tak jeho srozumitelnost a udržovatelnost, zvětšují robustnost kódu a usnadňují jeho další rozšiřování.

Velmi užitečná jsou vlákna v aplikacích s grafickým uživatelským rozhraním. Díky nim mohou aplikace poskytovat lepší odezvu pro uživatele a zpříjemňují tak i práci s těmito aplikacemi (kdy například nějaký složitý výpočet může běžet na pozadí, ale GUI bude stále reagovat na příkazy zadávané uživatelem). Serverové aplikace využívají vlákna pro lepší využívání sdílených prostředků a pro lepší výkon obsloužení více požadavků najednou. Můžeme říct, že všechny složitější aplikace jsou postaveny do jisté míry na využití vláken a jejich organizaci. Například JVM - garbage kolektor běží obvykle v jednom nebo ve více samostatně oddělených vláknech. [2]

2.1.2 Využití vláken na vícejádrových systémech

Další růst výkonnosti jedno-jádrových procesorů zpomalil. Vedl k vysokému příkonu a teplotě čipů, a to navzdory tomu, že se výrobcům mikroprocesorů daří neustále snižovat napájecí napětí a taktéž zmenšovat velikost aktivních i pasivních prvků na čipu. Více jader na čipu znamená větší výkonnost při stejném i nižším příkonu a přitom není nutné zvyšovat frekvenci a taktéž se nemusí dělat delší pipelina. Dnes se běžně setkáváme s dvou-jádrovými až osmi-jádrovými čipy a v budoucnu očekáváme desítky až stovky jader na čipu, zatímco výroba jedno-jádrových čipů je ve firmách omezena nebo zastavena úplně (v segmentu serverů a desktopů). Také aplikace dosud vždy běžely na další generaci procesorů rychleji. Tato éra již ale skončila, nově je třeba paralelní software.

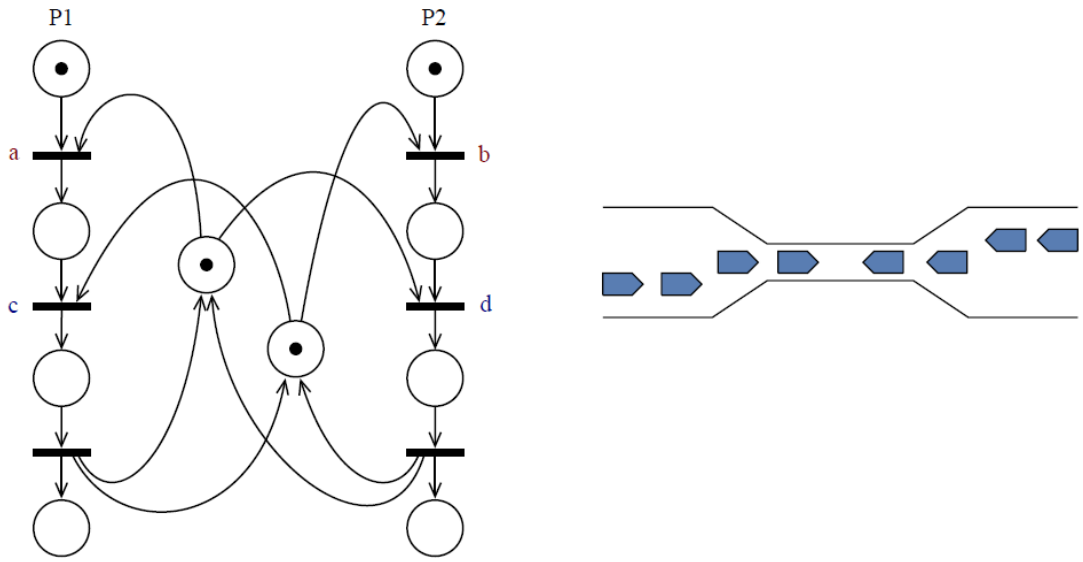
Zde se opět dostávají ke slovu vlákna, která představují základní jednotku pro plánování úloh na procesoru. Na vícejádrových systémech může běžet více aktivních vláken současně, což při správném návrhu umožňuje efektivní využití dostupných prostředků procesoru a zlepšuje propustnost (počet transakcí/s) u vícevláknových aplikací. [1]

2.1.3 Možná rizika použití vláken

Na jedné straně, kde nám vlákna usnadňují návrh a vývoj aplikace, tak na straně druhé použití vláken s sebou přináší mnohá bezpečnostní rizika (*safety hazards*), se kterýma jsme se dříve u jednovláknových programů nesetkali. Bezpečnostní riziko v tomto případě chápeme jako nekorektní či nechtěné chování programu, které může mít pro uživatele fatální následky. Při nesprávném použití vláken, bez využití jakýkoliv synchronizačních prostředků, je pak naprosto nepředvídatelné a někdy až překvapující pořadí jednotlivých operací, které program vykonává ve více vláknech paralelně. Další rizika souvisejí s živostí programu (*liveness hazards*), kdy může dojít ke zhoršení odezvy programu, jeho zamrznutí nebo ke snížení propustnosti, což má samozřejmě vliv i na celkový výkon vícevláknové aplikace. [2]

Patří sem rizika viz [5]:

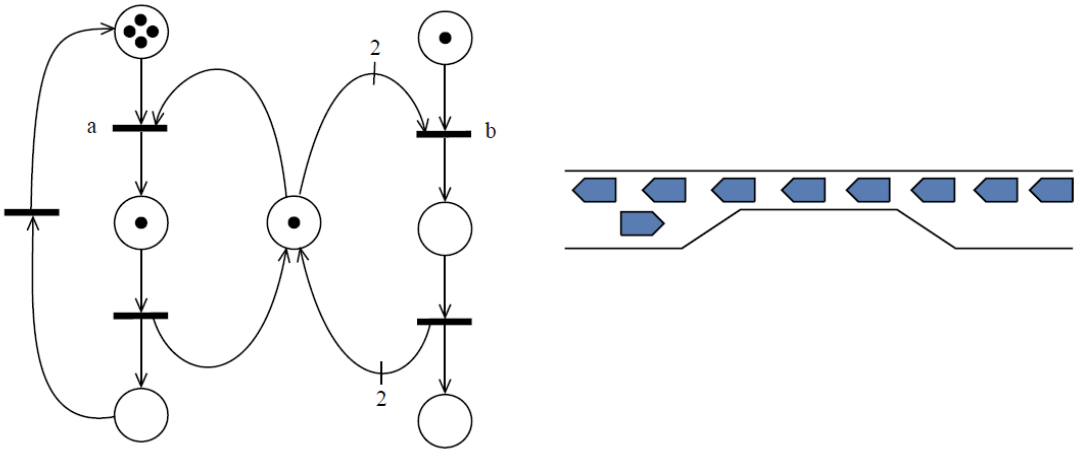
- uváznutí (deadlock) - rozumíme situaci, kdy každý ze skupiny procesů běžících v samostatných vláknech čeká na uvolnění zdroje s výlučným (omezeným) přístupem vlastněným nějakým procesem z dané skupiny



Obrázek 2.1: Uváznutí (deadlock)

- blokování (blocking) - rozumíme situaci, kdy proces žádá o vstup do kritické sekce, jejíž provádění jedním procesem vylučuje současné provádění této sekce ostatními procesy, musí čekat, přestože je kritická sekce volná a ani o žádnou z dané množiny kritických sekcí žádný další proces nežádá

- a hladovění (starvation) - rozumíme situaci, kdy proces čeká na podmínku, která nemusí nastat.



Obrázek 2.3: Hladovění (starvation)

- Zvláštním případem hladovění je tzv. livelock, kdy všechny procesy z určité množiny běží, ale provádějí jen omezený úsek kódu, který by mohly opustit jen tehdy, kdyby získaly zdroj vlastněný některým z procesů dané skupiny.

Pokud bezpečnost programu chápeme jako, že „nic zlého se nemůže stát“ a živost programu jako, že „něco požadovaného se nakonec stane“, tak výkonností programu pak rozumíme, že „něco požadovaného se vždy nakonec stane rychle“. Bezpečnost by měla být vždy na prvním místě a při návrhu naší aplikace bychom se měli řídit heslem: „Nejdřív to udělej tak, aby to bylo dobře a pak to udělej, aby to bylo rychlé, pokud už to není rychlé dostatečně“. Obecně taky platí, že dobrý návrh bývá mnohdy i efektivní návrh. [2]

2.2 Synchronizace při vícevláknovém přístupu

Vícevláknové zpracování probíhá asynchronně, což znamená, že vlákno probíhá nezávisle na ostatních vláknech. Tímto způsobem není vlákno závislé na provádění ostatních vláken. Jinak tomu je u procesů běžících synchronně. Ty jsou závislé jeden na druhém. To znamená, že jeden proces čeká na ukončení dalšího a teprve poté může běžet.

Někdy je však i provádění vlákna závislé na chodu jiného vlákna. Děje se tak například při sdílení zdrojů mezi vlákny. Hlavním problémem při sdílení jednoho zdroje dvěma nebo více vlákny je fakt, že v jednu chvíli může ke zdroji přistupovat jen jedno vlákno. Z toho plyne, že vlákna musí běžet synchronně a nikoliv asynchronně. Jednotlivá vlákna tak musejí čekat, dokud na ně nepříjde řada.

Vlákna se v Javě synchronizují za použití monitoru. Na monitor se díváme jako na objekt, který vláknu zpřístupňuje zdroj. Během jakéhokoliv jednoho časového intervalu může monitor používat pouze jedno vlákno. Říkáme, že pro daný časový interval vlákno vlastní monitor.

Vlákno smí monitor vlastnit jen tehdy, jestliže jej nevlastní žádné jiné vlákno. Pokud je monitor k dispozici a vlákno si jej přivlastní, získá tak exkluzivní přístup ke zdrojům přidruženým k monitoru. Pokud monitor k dispozici není, běh vlákna se pozastaví do té doby, dokud není monitor uvolněn. O takové situaci říkáme, že vlákno čeká na monitor.

Pro synchronizaci vláken lze využít dvou způsobů: synchronizovanou metodu nebo synchronizovaný příkaz. [3]

2.2.1 Synchronizovaná metoda

Všechny objekty v Javě mají monitor. Vlákno vstupuje do monitoru vždy, když zavolá metodu modifikovanou klíčovým slovem `synchronized`. O vláknu, které první volalo synchronizovanou metodu, se říká, že je uvnitř metody, a proto metodu vlastní spolu se zdroji používanými metodou. Další vlákno volající synchronizovaou metodu je pozastaveno do té doby, dokud první vlákno synchronizovanou metodu neopustí. Pokud je synchronizovanou metodou instanční metoda, aktivuje tato metoda zámek přidružený k instanci, která synchronizovanou metodu zavolala. V těle synchronizované metody je instance přístupná přes klíčové slovo `this`. V případě, že je synchronizovaná metoda statická, aktivuje zámek přidružený k objektu třídy, ve které se tato metoda definuje. [3]

```
synchronized void myFunciton() {
// Body of synchronized function
    ...
}
```

Příklad 2.1: Synchronizovaná metoda

2.2.2 Použití synchronizovaného příkazu

Synchronizovaná metoda představuje nejlepší způsob, jak omezit použití metody na jedno vlákno současně. Avšak čas od času se objeví situace, kdy metodu takto synchronizovat nelze. Například tehdy, když používáme třídu dodávanou jiným výrobcem. V takovém případě nemáme přístup k definici třídy, což zabraňuje použití klíčového slova `synchronized`.

Alternativou klíčového slova `synchronized` je synchronizovaný příkaz. Synchronizovaný příkaz obsahuje synchronizovaný blok, uvnitř kterého umisťujeme objekty a metody, které se mají synchronizovat. Volání metod obsažených v synchronizovaném bloku se uskuteční jen tehdy, když vlákno vstoupí do monitoru objektu.

I když metody uvnitř synchronizovaného bloku voláme, jejich deklarace se musí nacházet vně tohoto bloku. [3]

```
synchronized(list) {
// Body of synchronized block
    list.add(i);
}
```

Příklad 2.2: Ukázka použití synchronizovaného příkazu

3 Testování vícevláknových programů

Testování vícevláknových programů je založeno na podobných principech jako testování programů vykonávaných sekvenčně. Rozdíl je v tom, že vícevláknové programy mají v sobě určitou míru nedeterminismu, kterou sekvenční programy nemají. Důvodem nedeterminismu je větší počet možných interakcí a chybových stavů, které musí být ošetřeny. Ve vícevláknových programech je mnohem větší pravděpodobnost, že něco bude fungovat špatně, mnohé chyby jsou těžko předvídatelné a jejich vyskytování nemusí být pravidelné, ale může být závislé na určitých podmínkách. Proto testování vícevláknových programů musí být rozsáhlejší a o poznání déle trvající než jako je tomu u sekvenčních testů. Při vytváření testů můžeme využít nástroje pro návrh a analýzu některých stavů (například Petriho sítě a pod.). Tato kapitola vychází z [2].

3.1 Testy se zaměřením na bezpečnost

Vytvořit testy tak, aby dokázaly odhalit bezpečnostní chyby ve vícevláknových aplikacích je podobný problém, jako odpovědět na otázku, zda byla nejdřív slepice nebo vajíčko. Testovací programy jsou samy o sobě vícevláknové. Vytvořit dobrý testovací program vícevláknových aplikací může být mnohdy daleko složitější než vytvoření samotných aplikací.

Testování na bezpečnost se zaměřuje na ověřování, zda chování programu odpovídá jeho specifikaci. Vytvoření efektivních testů zaměřujících se na bezpečnost vyžaduje nalézt takové vlastnosti testované aplikace, u kterých lze předvídat vysokou pravděpodobnost, že u nich za určitých podmínek dojde k selhání, což si pak budeme moci snadno ověřit. Například si můžeme počítat kontrolní výpočty během testování a pokud vypočtená hodnota nebude odpovídat realitě, tak jsme zcela jistě objevili bezpečnostní chybu. Abychom si byli jistí, že testujeme to, co si myslíme, že testujeme, je důležité, aby naše kontrolní výpočty nebyly předvídatelné kompilátorem. Proto je dobrý nápad zvolit si pro testování náhodná data, protože chytré kompilátory by si mohly při špatném výběru testovacích dat snadno dopředu předpočítat výsledek. Obecně je lepší použít náhodná testovací data a naopak není vhodné používat pořád stejná data pro každý běh testu.

U vícevláknových aplikací je důležité dobře otestovat běh jednotlivých vláken, která jsou vykonávána současně. Vytvoření a odstartování jednoho vlákna však není jednoduchá operace. Pokud je běh vláken, která jsou postupně odstartována v cyklu, příliš krátký, může se v nejhorším případě stát, že budou vykonávána sekvenčně místo toho, aby běžela současně. I když nebudeme uvažovat nejhorší případ, stále je tady fakt, že některá vlákna jsou odstartována dříve než ostatní, což snižuje dobu, kdy vlákna běží současně. Tento problém můžeme zmírnit určením výchozího bodu, od kterého se započne testování. V Javě máme k dispozici cyklickou bariéru, na keré postupně vytvořená vlákna, předtím než můžou být současně vykonána, čekají.

Plánování procesů na vícejádrových systémech je nepředvítalné a tudíž i vzájemné interakce mezi vlákny ve vícevláknových aplikacích jsou velmi rozmanité. Abychom zvýšili šanci k odhalení nebezpečných přístupů ke sdíleným zdrojům, měli bychom testovat na systémech s menším počtem procesorů, na kterých vždy probíhá více aktivních vláken, než kolik je k dispozici výpočetních jednotek.

3.2 Testy se zaměřením na živost

Živost programu chápeme jako, že „něco požadovaného se nakonec stane“, tedy jinými slovy řečeno, že program dokončí požadovanou práci a nedojte k jeho zablokování nebo vyhladovění. Testování živosti programu obnáší své vlastní problémy. Během testování se provádí analýza pokroku programu, který je velice optížné vyčíslit. Je třeba si položit otázku, jak ověříme, zda-li je nějaká metoda blokující, nebo zda pouze neběží příliš pomalu. Jak otestujeme náš algoritmus, jestli náhodou nezpůsobuje uváznutí? Jak dlouho budeme čekat, než vyhodnotíme náš test, že skončil s chybou? To jsou otázky, na které si musíme odpovědět ještě předtím, než začneme vůbec testovat.

3.3 Testy zaměřené na výkonnost

Spolu s testy se zaměřením na živost souvisejí testy výkonnostní (*testing for performance*). Vskutku je velice vhodné v testech zaměřujících se na měření výkonnosti zahrnout také testy na základní funkcionalitu, abychom si tak mohli ověřit, jestli náhodou neprovádíme testování výkonnosti chybného kódu.

I když výkonnostní testy souvisejí s testy na základní funkcionalitu, mají odlišné cíle, které si musíme určit ještě před samotným zahájením testování. Definujeme klíčové metriky, které se k měření výkonnosti aplikace běžně používají. U metrik orientovaných na výkonnost nás zajímá efektivita práce se zdroji a propustnost (počet akcí, které lze dokončit v určitém časovém okamžiku). S definicí základních metrik si můžeme určit klíčové cíle výkonnosti (angl. *key performance targets*), které mají, kromě požadavků na efektivní práci se zdroji a propustnost, své požadavky například také na dostupnost a čas odezvy aplikace. Jelikož je každá aplikace unikátní, jsou unikátní také požadavky na její výkonnost. V důsledku unikátnosti každé aplikace neexistují žádné pevně dané hodnoty, se kterými bychom mohli naši aplikaci srovnávat, a tudíž dělat o její výkonnosti jakékoliv závěry bez definování klíčových cílů výkonnosti nemá smysl.

Testování provádíme na základě scénářů, které by měly v ideálním případě odpovídat reálnému použití aplikace uživateli. Předtím ale, než se pustíme do vytváření scénářů pro naše testy, si musíme být jistí, že je naše aplikace dostatečně stabilní pro výkonnostní testování. Primárním cílem výkonnostního testování není hledání chyb v aplikaci, proto by se nemělo stát, že čas vymezený na testování výkonu je spotřebován při opravě chyb. Aplikace je považována za stabilní, pracuje-li podle očekávání. Vytváření scénářů není vždy jednoduché, zatímco v některých případech jsou vhodné scenáře pro testování zřejmé z požadavků na aplikaci.

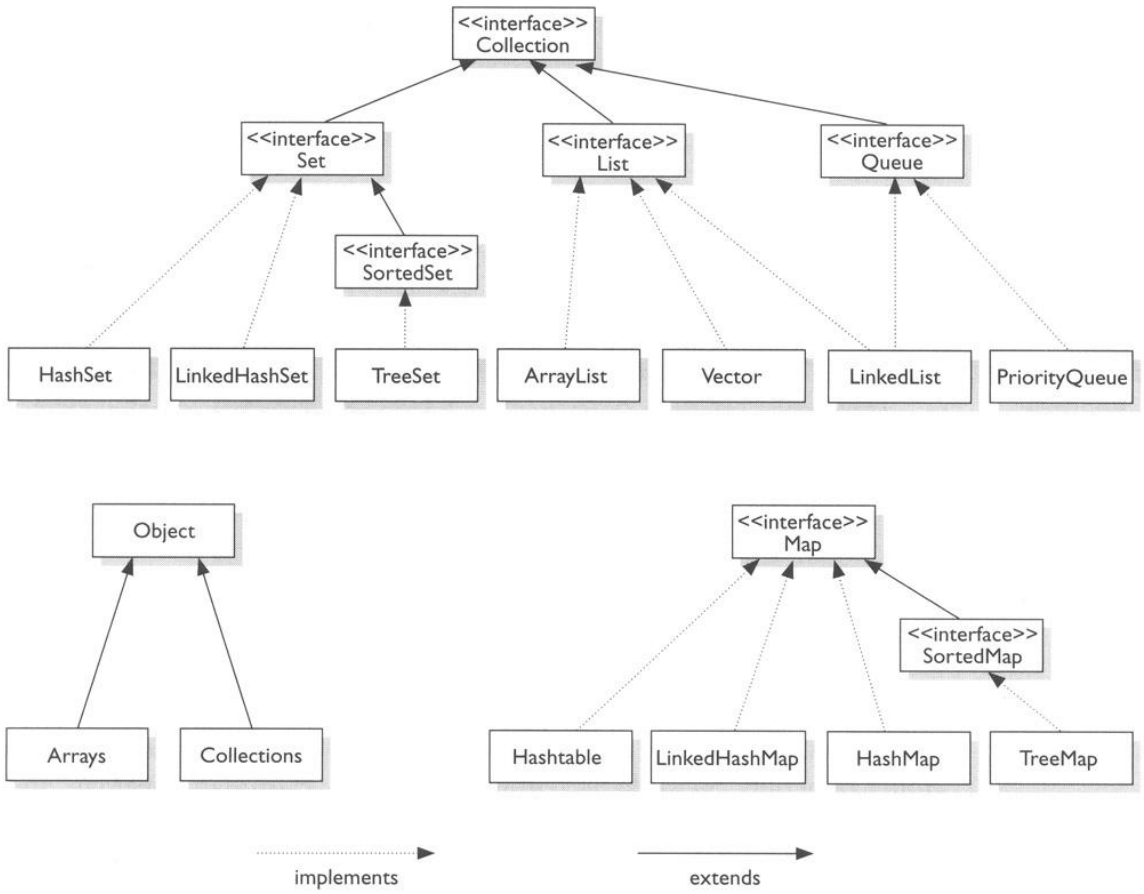
Při testování výkonnosti je rovněž důležité promyslet si, jak budeme postupovat při vybírání vhodné zátěže a její distribuci mezi jednotlivé funkce. Máme-li k dispozici předchozí verze aplikace, můžeme na základě jejich zkoumání získat věrohodná data o zátěži, se kterou je nutné počítat. Pokud předchozí verze aplikace k dispozici nemáme, tak nám nezbývá než zátěž odhadnout nebo ji získat od konkurence produkující podobný typ aplikace. Výkonnostní testy je možné provádět za relativně malé zátěže a slouží nám také mimojiné k porovnávání výkonu mezi jednotlivými verzemi aplikace.

Porovnávání výkonnosti různých objektů se stejnou funkcionalitou se označuje pod anglickým pojmem *benchmarking*. Provádí se testování různých funkcí objektu a naměřené výsledky se porovnávají s výsledky měření u jiných objektů. Benchmarking se často používá pro marketingové účely při propagaci nových produktů.

4 Java Collection Framework

Java Collection Framework [6] tvoří jednotný rámec pro reprezentaci a manipulaci s kolekcemi. Umožňuje nám nezávisle manipulovat s jejich vnitřní reprezentací, snižuje nutné úsilí programátora vynaložené při práci s těmito kolekcemi a zároveň zlepšuje celkový výkon naší aplikace. Umoňuje vzájemnou interoperabilitu mezi nezávislými rohraními, snižuje také úsilí při vytváření a učení se novým rozhraním a podporuje znovuvyužití již hotových implementací.

Framework pro Javu ve verzi 6.0 obsahuje implementaci čtrnácti rozhraní pro různé typy kolekcí, jako jsou množina (angl. *Set*), seznam (angl. *List*) a mapa (angl. *Map*). Tyto rozhraní utváří základ celého frameworku. Kromě základních implementací se zde nachází také implementace navržené pro speciální případy, kdy potřebujeme dosáhnout nějakého nestandardního chování, specifických výkonnostních charakteristik anebo např. stanovit nějaká omezení při používání dané kolekce. Ve vícevláknových aplikacích můžeme využít implementací navržených pro účely souběžného přístupu (concurrent implementations) nebo zapouzdřených implementací (wrapper implementations) poskytující navíc synchronizační nástroje.



Obrázek 4.1: Základní struktura kolekcí ve standardní knihovně

Ve standardní knihovně Javy jsou základní kolekce definovány v balíčku `java.util`. Jak můžeme vidět na obrázku 4.1 [9], kořenovým rozhraním v hierarchii kolekcí je rozhraní `Collection`, které rozšiřuje rozhraní `Iterable` umožňující používat kolekce v příkazu „`foreach`“. Rozhraní `Collection` shromažďuje implementace algoritmů pro práci s kolekcemi, které jsou obecně navrženy tak, aby bez ohledu na implementaci kolekce zajišťovaly minimální

operační složitost. Od tohoto kořenového rozhraní jsou odvozená rozhraní `Set`, `List`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `BlockingQueue`, `BlockingDeque`. Zbývajících pět rozhraní pro kolekce `Map`, `SortedMap`, `NavigableMap`, `ConcurrentMap` a `ConcurrentNavigableMap` řadíme do kategorie tzv. asociativních kolekcí („mapované kolekce“) obsahující dvojice klíč-hodnota. Tato rozhraní nejsou odvozená od rozhraní `Collection`, obsahují však operace, tzv. „*collection-view operations*“ (metody pro zobrazení rozhraní `Collection`), které nám umožňují s němi zacházet jako s kolekcemi. Jsou to metody `keySet()`, `entrySet()` a `values()`. Jednotlivá rozhraní jsou tvořena specifikacemi prototypů metod, které podporují. Hodně metod pro práci s kolekcemi jsou označeny jako volitelné (angl. *optional*) a některé kolekce tyto metody nemusí podporovat. V případě, kdy použijeme nepodporovanou metodu, nám kolekce vrátí vyjímku `UnsupportedOperationException`. Jaké operace daná kolekce podporuje a nepodporuje lze dohledat v dokumentaci k dané kolekci.

Třídy implementující jednotlivá rozhraní kolekcí mají typický název ve tvaru *<styl implementace><název rozhraní>*.

		Implementations				
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Interfaces	**Set**	`HashSet`		`TreeSet`		`LinkedHashSet`
	List		`ArrayList`		`LinkedList`	
	Deque		`ArrayDeque`		`LinkedList`	
	Map	`HashMap`		`TreeMap`		`LinkedHashMap`

Tabulka 4.1:Implementace základních rozhraní v Java Collection Framework 6.0

Tyto základní implementace kolekcí uvedené v tabulce 4.1[6] jsou určeny pro obecné použití. Vyznačují se tím, že podporují všechny operace nad těmito kolekcemi, které jsou označeny jako volitelné. Jsou to například metody modifikační jako jsou `add()`, `remove()` nebo `clear()`. Takové kolekce potom nesou přívlastek modifikovatelné (angl. *modifiable*). Výše uvedené kolekce nemají rovněž žádná omezení ohledně prvků, které mohou obsahovat. Neobsahují žádné synchronizační prostředky, a tedy nepodporují ani vícevláknový přístup k těmto kolekcím. Pro vícevláknový přístup můžeme využít třídu `Collections`, která obsahuje statické metody `synchronizedMap()` a `synchronizedList()` vracející zapouzdřenou implementaci kolekcí, které jsou již bezpečné z hlediska použití při souběžném přístupu z více vláken najednou.

Hlavním cílem rámce Java Collection Framework bylo vytvořit pro programátory implementace základních kolekcí, které jsou optimalizované a dosahují tedy vysokého výkonu. Poskytují pro různé typy kolekcí podobný způsob zacházení. Rozhraní zde představují abstraktní pohled na danou kolekci a umožňují manipulaci s danou kolekcí nezávisle na její vnitřní reprezentaci. V objektově orientovaných jazycích je rozhraní schéma, které deklaruje seznam metod bez implementace. Definice těchto metod jsou pak uvedeny v konkrétních třídách pro danou kolekci, které implementují dané rozhraní. Dále Java Collection Framework definuje několik rozhraní a tříd pro mapy, asociativní kolekce, které nejsou v pravém slova smyslu kolekce, ale jsou s nimi plně integrovány.

4.1 Rozhraní Set

Rozhraní `Set`[4], neboli česky množina zdánlivě nenabízí kromě metod zděděných z rozhraní `Collection` nic navíc. Rozdíl oproti obecné kolekci zde však je. Kontejner implementující rozhraní `Set` odpovídá matematickému pojmu množina, tedy uspořádaná kolekce objektů, kde se každý prvek vyskytuje právě jednou. Kromě toho může `Set` obsahovat nejvýše jeden nulový (null) odkaz. Prvky nemají určenou žádnou polohu v množině, existuje však i seřazená varianta, která implementuje rozhraní `SortedSet`. Množiny se nejčastěji používají pro ukládání prvků, kdy nejsou povoleny duplicitní prvky, a kdy nám nezáleží na jejich pořadí.

4.1.1 HashSet

Nejlepší implementací rozhraní `Set` poskytuje kolekce typu `HashSet`[4, 6], která pracuje na principu tzv. *hašování* vkládaných prvků. To nám nezaručuje uspořádání prvků v množině, obvzlášť pak, ani že pořadí prvků zůstane konstatní. Avšak práce s touto množinou je velice rychlá, většina základních operací jako jsou `add()`, `remove()`, `contains()` a `size()` probíhá v konstantním čase. Procházení touto množinou vyžaduje čas úměrný její velikosti (kolik prvků daná množina obsahuje) a kapacitě této množiny. Proto je pro její optimální výkon důležité správně zvolit její výchozí kapacitu. Zvolíme-li tedy příliš vysokou počáteční kapacitu, zbytečně pak plýtváme prostorem i časem. Na druhou stranu výběr příliš nízké počáteční kapacity také prodlužuje operace, protože je nutné datovou strukturu kopírovat pokaždé, když musí svou kapacitu zvýšit. Pokud počáteční kapacitu neurčíme, použije se výchozí hodnota 16.

4.1.2 LinkedHashSet

Jen o něco málo efektivní `LinkedHashSet`[4, 6] je budován na stejném principu jako `HashSet`. Kromě jeho vlastností přináší schopnost procházení iterátorem v uspořádání daném pořadím vkládáním prvků (od nejdříve vložených po nejpozději vložené prvky). Iterátor nad běžným `HashSet`em totiž prochází prvky v „chaotickém“ pořadí.

4.1.3 TreeSet

Množinu se stromovou strukturou, `TreeSet`[4, 6] použijeme pro práci se seřazenými prvky. Protože je to seřazená množina, implementuje rozhraní `SortedSet`, které zaručuje uspořádanost prvků vkládaných do množiny. Je mocnější, ale obvykle méně efektivní - operace jsou většinou pomalejší než u `HashSet`, provádějí se v logaritmickém čase. Tím, že `TreeSet` implementuje rozhraní `SortedSet`, poskytuje navíc, kromě normálních operací kolekce typu `Set`, další operace umožňující zobrazení libovolného rozsahu seřazené množiny (`subSet()`, `headSet()`, `tailSet()`), operace, které vrátí první nebo poslední prvek (`first()`, `last()`) a přístup k objektu typu `Comparator`, použitý při řazení, pokud tento objekt existuje. Operace, které `TreeSet` dědí od rozhraní `Set`, se chovají stejně se dvěma vyjímkami:

- Objekt typu `Iterator` vrácený operací `iterator` prochází seřazenou množinu v pořadí prvků.
- Pole vrácené metodou `toArray()` obsahuje prvky seřazené množiny v jejich pořadí.

Specialitou je také metoda `toString()`, která na platformě Java vrací seřazený řetězec ve správném pořadí, i když to rozhraní `SortedSet` nezaručuje.

4.2 Rozhraní Map

Rozhraní `Map`[4, 6], které je rovněž součástí platformy Java Collection Framework, jak už jsem zmínil výše, není odvozeno od rozhraní `Collection`. Kolekce typu `Map` jsou také někdy nazývány termínem slovník. Neukládáme do ní samotné prvky, ale dvojice prvků. V této dvojici je vždy jeden prvek označen jako klíč a druhý jako hodnota. Mapy slouží především k ukládání a následnému vybírání hodnot. Kdykoliv chceme získat hodnotu uloženou v mapě, zadáme klíč a mapa nám vrátí jemu přiřazenou hodnotu. Je to obdobné, jako když hledáme v překladovém slovníku: zadáme klíč = slovíčko v jednom jazyku a slovník nám vrátí hodnotu = slovíčko v jiném jazyku. Termín mapa se pro tyto druhy kolekcí používá proto, že *mapuje* klíče na jejich hodnoty. Mapa nemůže obsahovat duplicitní klíče a každý klíč lze mapovat nejvýše na jednu hodnotu. Modeluje tak abstrakci matematické funkce.

Základní operace tohoto rozhraní jsou `put()` pro přidávání prvků, `get()`, která nám vrátí zmíněnou hodnotu uloženou pod daným klíčem, metoda `remove()` odebírající dvojici klíč-hodnota z kolekce po zadání klíče, dále to jsou dotazy `containsKey()` a `containsValue()`, zda se daný klíč nebo hodnota nacházejí v kolekci, dotaz `size()`, který vrací velikost kolekce a dotaz `isEmpty()`, zda je daná kolekce prázdná. Kromě základních operací, nám `Map` nabízí metody zobrazení rozhraní `Collection`, které umožňují zobrazit objekty typu `Map` jako objekty typu `Collection`. Jsou to metody `keySet()`, která vrací objekt typu `Set` s klíči, které jsou uloženy v objektu typu `Map`, `values()` vracející objekt typu `Collection` s hodnotami, které jsou součástí objektu typu `Map` (tento objekt typu `Collection` není typu `Set`, protože lze mapovat více klíčů na stejnou hodnotu) a metoda `entrySet()`, která vrací objekt typu `Set` s dvojicemi klíčů a hodnot, které josu součástí objektu typu `Map`. Rozhraní `Map` poskytuje malé vnořené rozhaní s názvem `Map.Entry` s typy prvků v příslušném objektu typu `Set`. Zobrazení rozhraní typu `Collection` představují jediný způsob, jak procházet objekty typu `Map`. V případě zobrazení `entrySet` je také možné změnit hodnotu přidruženou ke klíči voláním metody `setValue()` objektu typu `Map.Entry`. Jedná se o jediný bezpečný způsob, jak upravit objekt typu `Map` během procházení (předpokládá se, že objekt typu `Map` podporuje úpravy hodnot). Zobrazení rozhraní Collection také podporují odebírání prvků ve všech možných variantách jako jsou `remove()`, `removeAll()`, `retainAll()` a `clear()` (znovu se předpokládá, že podpůrný objekt typu Map je kompatibilní s odebráním prvků).

Podobný objekt typu `Map` je tzv. vícenásobná mapa (angl. *multimap*), která nám dovoluje mapovat každý klíč na více hodnot. Platforma Java Collection Framework ale neobsahuje žádné rozhraní vícenásobných map, protože se nepoužívají příliš často a je je poměrně jednoduché implementovat jako objekt typu `Map`, jehož hodnotami jsou instance rozhraní `List`.

4.2.1 HashMap

Implementace s *hašovací* tabulkou, `HashMap`[6] přesně odpovídá chování a výkonu implementace `HashSet`. Velmi efektivní, každý ukládaný objekt se převede na číslo - *haš kód*, které určí jeho umístění v tabulce. Hledání prvku v tabulce probíhá analogicky, kdy je prvek uložen na pozici dané *haš kódem*. Mají-li se dva objekty umístit na stejnou pozici, vznikne kolize, kterou je třeba nějak řešit. Na výkon této kolekce mají vliv dva parametry. Jsou to podobně jako u `HashSet` výchozí kapacita a tzv. *load faktor*. *Load faktor*, nebo-li česky faktor „*plnění*“ určuje, při jakém stupni zaplnění tabulky je vhodné tabulku zvětšit, protože už je příliš vysoká pravděpodobnost kolizí. Čím je menší faktor *plnění*, tím je větší šance, že nám při dobře navržené *hašovací* funkci, nebude pro několik prvků vycházet stejné místo v tabulce. Dokumentace říká, že rozumný kompromis mezi rychlým vyhledáváním a malou spotřebou paměti je hodnota *plnění* 0.75, což je také implicitní hodnota.

`HashMap` poskyuje implementaci všech volitelných operací z rozhraní `Map`. Proto vykazuje velice časté použití ve všech případech, kdy nepotřebujeme mít seřazené klíče. Umožňuje prázdné (null) klíče a hodnoty. Kolekce není synchronizovaná, takže při použití s vícevláknovým přístupem musíme použít externí synchronizační prostředky.

4.2.2 TreeMap

Implementace `TreeMap`[6] poskytuje oproti `HashMap` také operace z rozhraní `SortedMap`. Toto rozhraní představuje přesnou analogii rozhraní `SortedSet` (kapitola 4.1.3) v rámci rozhraní `Map`. `TreeMap` použijeme v případě, pokud potřebujeme operace z rozhraní `SortedMap` nebo když potřebujeme procházet kolekci pomocí zobrazení rozhraní `Collection` s řazením podle klíčů. Uspořádanost klíčů je zde zaručena jejich ukládáním do tzv. „*černobílého stromu*“. Tato vyhledávácí struktura zaručuje efektivní ukládání, dotazování i rušení dvojic klíč-hodnota.

Zkonstruovat kolekci `TreeMap` můžeme třemi možnými způsoby. Použijeme-li prázdný konstruktor `TreeMap()`, vznikne mapa, v níž jsou prvky řazeny v přirozeném pořadí (to je dané chováním metody `compareTo()`). Konstruktor, jemuž poskytneme komparátor - `TreeMap(Comparator c)` vytvoří mapu s prvky řazenými tímto komparátorem, což nám poskytuje větší flexibilitu, například možnost jednoduše dosáhnout obráceného (sestupného) uspořádání klíčů. Třetí varianta umožňuje použít konstruktor přebírající (neuspořádanou) mapu - `TreeMap(Map m)`, který vytvoří novou uspořádanou mapu se stejnými dvojicemi, jako měla výchozí mapa `m`. Bohužel nelze pvky v nové mapě uspořádavat jinak než v přirozeném pořadí. Nové mapě totiž nelze při konstrukci vnutit komparátor.

4.3 Rozhraní List

Rozhraní `List`[4, 6] je rozšířením (potomkem) rozhraní `Collection`. `List`, nebo-li česky seznam je uspořádanou kolekcí, která se někdy označuje též jako *sekvence*. Uživatel kolekce `List` má zpravidla přesnou kontrolu nad místem vložení každého prvku seznamu a může přistupovat k prvkům pomocí jejich celočíselného indexu. Narozdíl od množiny `List` povoluje duplicitní prvky.

Kromě operací zděděných od rozhraní `Collection` zarhnuje také operace využívající manipulace s prvky na základě jejich pozice v seznamu. Mezi tyto operace patří metody jako jsou `get(int index)`, která vrátí prvek na pozici `index` (pokud takový prvek není, je vyvolána vyjímka `java.lang.IndexOutOfBoundsException`), `set(int index, Object element)`, která na pozici `index` nahradí dosud uložený prvek prvkem `element` a původní prvek vrátí jako návratovou hodnotu, `add (int index, Object element)`, která podobně jako `set` s rozdílem, že nový prvek `element` na pozici `index` vkládá a předchozí prvek jím nenahrazuje, `addAll(int index, Collection c)`, která vloží všechny prvky z kolekce `c` od pozice `index` počínaje, a `remove(int index)` vyřaďující prvek na pozici `index`. Rozšířením oproti původnímu rozhraní jsou i metody, které vyhledají v seznamu určený prvek a vrátí jeho číselnou pozici. Jsou to metody `indexOf(Object o)` a `lastIndexOf(Object o)`. K dispozici je nám také nový *seznamový iterátor* `listIterator()`. Od běžného iterátoru nad kolekcí se liší tím, že umožňuje procházet prvky v přesně daném pořadí a to i zpět. Navíc je možné začít procházet až od určité pozice v seznamu - `listIterator(int index)`. Pro zobrazení rozsahu (angl. *range-view*) seznamu můžeme využít metodu `subList(int fromIndex, int toIndex)`, která nám vrátí podseznam začínající na pozici `fromIndex` a končící prvkem předcházejícím pozici `toIndex`. Pokud jsou pozice `fromIndex` nebo `toIndex` mimo meze, vyvolá se běhová vyjímka `java.lang.IndexOutOfBoundsException`. Musíme si však dávat pozor při práci s touto metodou, protože veškeré změny v tomto zobrazení se odrážejí i v původním seznamu. Vrácený podseznam tedy není kopií z původního seznamu, nýbrž jde jen o „*jiný pohled*“ na táž data. Proto lze rozhodně doporučit používání tohoto zobrazení vraceného z metody *`subList`* pouze jako dočasný objekt k provedení jedné nebo několika operací nad původním seznamem.

4.3.1 ArrayList

`ArrayList`[4, 6] je kolekce implementující přímo rozhraní `List` (a řadu dalších rozrhaní jako jsou například `Iterable` nebo `RandomAccess`). Již svým názvem napovídá, že bude mít hodně společného s polem. Ve skutečnosti se jedná o *kontejner* postavený nad běžným polem, který umožňuje to, co klasické pole neumí. Mezi tyto přednosti patří možnost měnit (zvyšovat) počet uložitelných prvků, v kládat nové prvky mezi již uložené atd.

`ArrayList` má podobně jako některé další kolekce svou kapacitu - počet prvků, který je možné po sobě do kolekce vložit, aniž by muselo dojít ke zvětšování bázového pole, nad nímž je `ArrayList` postaven. Pokud dopředu víme, že v kolekci bude mnoho prvků, je lepší rovnou při konstrukci `ArrayList`u nastavit větší výchozí kapacitu konstruktorem `ArrayList(int vychoziKapacita)`. V průběhu práce je pak možné využít metodu `ensureCapacity(int minKapacita)`, která podle potřeby zajistí okamžité zvětšení prostoru v *kontejneru* tak, aby pojal alespoň minimální kapacitu určenou parametrem `minKapacita`. Dobré zvládnutí této techniky může mít v jistých případech značný vliv na výkon.

Většina operací, kde se využívá přístup přes indes, probíhá zde v konstantním čase. Přidávání nebo modifikace uvnitř posloupnosti v čase lineárním. Tuto kolekci použijeme, když bychom jinak použili běžné pole, ale potřebujeme měnit délku. Nehodí se v případech, kdy velmi často dochází k přidávání prvků na začátek nebo provádění změn uvnitř seznamu.

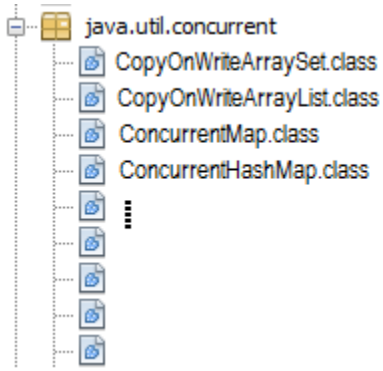
4.3.2 LinkedList

Je též implementací rozhraní `List`, poskytuje všechny známé operace z tohoto rozhraní a nemá žádná omezení na vkládané prvky (povoluje vkládat i nulové prvky `null`). `LinkedList`[4, 6] disponuje navíc metodami pro přímé vkládání a odebírání prvků na začátku a konci seznamu. Díky tomu je možné `LinkedList` chápat jako zásobník (angl. *stack*), frontu (angl. *queue*) nebo oboustrannou frontu (angl. *double-ended queue*). Tyto vlastnosti jsou zajištěny implementací rozhraní `Queue` a `Deque`.

Operace pro přídávání a mazání probíhají v konstantním čase, zatímco u kolekce typu `ArrayList` probíhají v čase lineárním. Proto je vodné použití této kolekce v případech s častými modifikacemi na jiných místech, než je konec seznamu. Platíme však vysokou výkonností cenu při pozičním přístupu, kdy je nutný lineární čas. Při sekvenčním přístupu je obecně mnohem efektivnější používat iterátory.

4.4 Balíček java.util.concurrent

Balíček `java.util.concurrent`[10] se vyskytuje v Javě až od verze 5.0. Přináší nové, velmi efektivní synchronizační prostředky pro programátory vícevláknových aplikací. Tyto prostředky jsou rozděleny do kategorií. Jednu z nich tvoří mimojiné paralelní kolekce (angl. *concurrent collections*), které jsou přímo navrženy tak, že poporují současný přístup z více vláken najednou. Paralelní kolekce představují již hotová řešení, která lze jednoduše použít, čímž nám usnadňují práci při vytváření vícevláknových programů. Obrázek 4.2 představuje výřez z balíčku a ukazuje, jaké kolekce nám tento balíček může například nabídnout.



Obrázek 4.2: Některé třídy kolekcí z balíčku java.util.concurrent

Předpona „*Concurrent*“ v názvu kolekcí znamená v češtině souběžný, současný nebo taky paralelní. To je taky přesný rozdíl oproti synchronizovaným kolekcím jako je například `synchronizedMap`. Synchronizované kolekce umožňují ve stejný okamžik přístup pouze z jednoho vlákna. Paralelní kolekce se od těch synchronizovaných liší i tím, že poskytují bezpečné chování při vícevláknovém přístupu pro sekvence operací, kdy výsledek jedné operace záleží na výsledku předcházející operace. Je zde definovaná relace „*nastává-před*“ mezi operací přidání do kolekce a operací čtení nebo odstranění z kolekce.

4.4.1 ConcurrentHashMap

`ConcurrentHashMap`[10] z balíčku `java.util.concurrent` je alternativa pro kolekce typu `Hashtable` nebo `synchronizedMap`. Používá úplně jinou strategii zamykání kolekce, která přináší mnohem lepší možnost vícevláknového přístupu. Místo synchronizace každé metody běžným zámkem, což vylučuje souběžný přístup z více vláken ve stejný okamžik, používá mechanismus založený na jemnějším zamykání, který si taky můžeme představit jako prokládání zámků kolekcí (angl. se tento mechanismus nazývá *lock striping*) a který umožňuje lepší stupeň sdíleného přístupu. Je možné nastavit úroveň souběžnosti (angl. *concurrency level*), což je vnitřní segmentace. Tím, že stupeň segmentace určíme, říkáme vlastně, kolik vláken může současně měnit data v mapě. To má za následek neblokující chování pro vlákna přistupující k této kolekci.

Jak bychom asi očekávali, poskytuje všechny operace z rozhraní `Map`, taktéž implementuje i rozhraní `ConcurrentMap`, které rozšiřuje původní rozhraní o atomické operace *put-if-absent*, *remove-if-equal* a *replace-if-equal*. Při procházení kolekcí máme k dispozici iterátor, který nikdy nevrací vyjímku `ConcurrentModificationException`. Během iterace nedochází k uzamčení kolekce, ale pracujeme s iterátorem, který reflektuje stav mapy v okamžiku jeho vytvoření a nebere tak v potaz změny, které v kolekci probíhají během existence iterátoru. Podobně i ostatní operace pracující s celou kolekcí jako jsou `size()` nebo `isEmpty()` jsou zde kvůli konkuretní povaze této kolekce lehce oslabeny a výsledek těchto operací nemusí být v danou chvíli aktuální, ale bereme ho pouze jenom jako odhad blížícímu se k přesnému výsledku. Ve vícevláknových aplikacích je toto chování tolerováno, protože se očekává, že paralelní kolekce budou nepřetržitě měnit svůj obsah. Více je zde kladen důraz na výkon operací `get`, `put`, `containsKey` a `remove`.

4.4.2 CopyOnWriteArrayList

Náhrada za synchronizovanou kolekci seznam je v balíčku `java.util.concurrent` `CopyOnWriteArrayList`[10]. Přináší podobně jako `ConcurrentHashMap` lepší podporu pro současný vícevláknový přístup. Bezpečnost z hlediska vícevláknového přístupu spočívá v tom, že pracujeme s neměnným polem, takže při čtení nemůže dojík k jeho modifikaci a tudíž není nutná žádná synchronizace. Operace zápisu fungují na principu vytváření nové kopie pole a její aktualizaci. Iterátory fungují obdobně jako u předešlé kolekce a i zde je zaručeno, že nikdy nezpůsobí vyjímku typu `ConcurrentModificationException`.

4.5 Vybrané kolekce z balíku Javolution

Projekt `Javolution`[7] vznikl v roce 2001, dříve označován zkratkou J.A.D.E. (Java Addition to Default Environment) a jeho autorem je Jean-Marie Dautelle. Tento projekt je stále vyvíjen skupinou programátorů a spadá pod BSD licenci, což je licence pro svobodný software.

Javo(So)lution přináší vylepšená řešení základních tříd ze standardní knihovny Javy (balíčky `util/ lang/ text/ io/` a `xml/`) a zaměřuje se hlavně na rychlost aplikací a lepší předvídavost časové náročnosti. Využití nalezneme například v *real-time* systémech nebo také vestavěných systémech. Balíček `javolution.util` poskytuje několik velice rychlých implementací kolekcí, které můžou být ve většině případů použity místo běžných kolekcí.

4.5.1 FastMap

`FastMap`[8] z balíčku `javolution.util` dodržuje implementaci rozhraní `Map`, takže může sloužit jako jednoduchá náhrada za `HashMap` nebo také `LinkedHashMap`. Při konstrukci lze aktivovat příznak pro sdílení a využít tak tuto kolekci i jako altenativu pro kolekci `ConcurrentHashMap`. Příznak pro sdílení nastavíme pomocí metody `shared()`, která nám zaručuje podporu vícevláknového přístupu.

`FastMap` se chlubí plynulým zvyšováním kapacity. Nedochází nikdy k přehashování celé mapy, ale v případě, že počet záznamu překročí její kapacitu, alokuje se pro nové záznamy nové místo v paměti, přičemž staré záznamy se nikam nepřesouvávají. Záznamy se ukládají v pořadí podle klíčů, v jakém byly do mapy vloženy (podobně jak je to u kolekce `LinkedHashMap`). Tato kolekce implementuje i rozhraní `Reusable` a `Realtime`. Udržuje vnitřní úložiště pro záznamy a pokud je nějaký záznam vymazán z mapy, je toto úložiště automaticky obnoveno.

4.5.2 FastList

`FastList`[8], dvousměrný vázaný seznam představuje rozšíření implementace rychlých kolekcí z balíčku `javolution.util`. Je potomkem třídy `FastCollection`, která se vyznačuje zejména rychlým oboustranným procházením kolekce, aniž by se vytvářel nový objekt iterátor.

Poskytuje všechny operace, které bychom mohli očekávat u dvousměrného vázaného seznamu. Vkládání prvků na začátek seznamu a na konec (které je rychlejší). Operace přístupu procházejí seznamem od začátku nebo od konce seznamu podle toho, ke které straně je specifikovaný index blíže. Při náhodném přístupu je čtení výrazně zrychleno rozdělením seznamu na menší podseznamy.

Stejně jako `FastMap`, tak i `FastList` implementuje rozhraní `Reusable` a `Realtime` a umožňuje tedy efektivní práci s alokovanou pamětí. Udržuje vnitřní úložiště pro uzly seznamu, které je, po jejich vymazání, automaticky obnoveno.

Implementace `FastListu` umožňuje překrývání metody `newNode()`, která může vrátit uživateli jeho vlastní implementaci objektu `FastList.Node` (s dodatečnými poli například).

Kolekce neposkytuje metodu `shared()` a tedy ji nelze označit jako sdílenou pro souběžný přístup z více vláken najednou. Bezpečné chování při vícevláknovém přístupu musíme zajistit pomocí explicitní synchronizace, například pomocí metody `synchronizedList()`.

5 Návrh vhodných zátěžových testů pro měření výkonnosti datových kolekcí

Tato kapitola se zabývá návrhem vhodných testů, které se zaměřují na výkonnost základních operací datových kolekcí při souběžném vícevláknovém přístupu. Předmětem zkoumání jsou zde kolekce z rámce pro práci s kolekcemi `Java Framework Collection`, dále paralelní kolekce z balíčku `java.util.concurrent` a vybrané implementace rychlých kolekcí vyvinuté externí skupinou programátorů z projektu `Javolution`. Jedna se o kolekce implementující rozhraní `Map` a rozhraní `List`.

5.1 Testy kolekcí implementující rozhraní Map

Rozhraní `Map` poskytuje základní operace pro vkládání a pro čtení uvedené v kapitole *4.2 Rozhraní Map*. Při návrhu jsem si musel nejdříve rozmyslet, jak a za jakých okolností budu základní operace testovat. Vytvořil jsem si sadu různých testů, kterou vyjadřuje tabulka 6.1. Třídy představující implementace kolekcí, které se zde budou testovat, jsou to `HashMap`, `ConcurrentHashMap`, `FastMap` a `TreeMap`. Při konstrukci těchto kolekcí uvedeme jako datový typ pro dvojice klíč-hodnota `Integer` jak pro klíč, tak i pro hodnotu.

Výkonnost jednotlivých operací se bude testovat pro různé sekvence vkládaných hodnot celých čísel. Pro zátěžové testy jsem vybral vzestupnou seřazenou posloupnost od nuly po číslo `n`, které určuje maximální velikost kolekce. Opakem vzestupné sekvence je sestupná sekvence čísel od `n` po nulu. Pak zde budeme mít jednu náročnou sekvenci, jíž si pojmenovávám jako „*hard*“, a která bude testovat vkládání nebo čtení prvků v pořadí `0, n, 1, n-1`, a tak dále až po `n/2` nebo `n/2-1`. Poslední sekvencí, kterou jsem zvolil a která nám určitě taky přinese zajímavé výsledky, je sekvence náhodná.

Poslední dvě sady testů se zaměřují na zvyšování velikosti kolekce a počtu vláken, které k ní současně přistupují. Minimální velikost kolekce, pro kterou budeme testovat je 100 uložených prvků. Postupně budeme velikost zvyšovat násobky desíti až do maximální hodnoty 100 000. Rozdílný výkon kolekce budeme pravděpodobně sledovat i při přístupu ke kolekci pouze z jednoho vlákna oproti přístupu vícenásobného. Budeme testovat až pro 24 vláken přistupujících současně ke kolekci.

INTERFACE:	MAP				

CLASSES:	`HashMap<K,V>`
	`ConcurrentHashMap<K,V>`
	`FastMap<K,V>`
	`TreeMap<K,V>`				

METHODS:	put()	get()	containsKey()	containsValue()	remove()
TYPE (K,V):	Integer, Integer				
INTEGER SEQUENCES:	Ascending (0..n) Descending (n..0) Hard (0, n, 1, n-1, ... n/2, n/2-1) Random sequence				
SIZE (n):	n = 100 n = 1 000 n = 10 000 n = 100 000				
NUMBER OF THREADS:	1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24				

Tabulka 6.1: Návrh zátěžových testů pro kolekce implementující rozhraní Map

5.2 Testy kolekcí implementující rozhraní List

Podobně jako při vytváření návrhu testů kolekcí implementující rozhraní `Map`, jsem navrhnul i testy pro kolekce implementující rozhraní `List`, které uživatelům poskytuje podobné operace (viz kapitola *4.3 Rozhraní List*). Předmětem testování budou kolekce typu `ArrayList`, jeho paralelní obdoba `CopyOnWriteArrayList`, lineárně vázaný seznam `LinkedList` a `FastList`.

INTERFACE:	LIST			

CLASSES:	`ArrayList<E>`
	`CopyOnWriteArrayList<E>`
	`LinkedList<E>`
	`FastList<E>`			

METHODS:	add()	get()	set()	remove()
TYPE (E):	Integer			
INTEGER SEQUENCES:	Ascending (0..n) Descending (n..0) Hard (0, n, 1, n-1, ... n/2, n/2-1) Random sequence			
SIZE (n):	n = 100 n = 1 000 n = 10 000 n = 100 000			
NUMBER OF THREADS:	1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24			

Tabulka 6.2: Návrh zátěžových testů pro kolekce implementující rozhraní List

Celkový návrh zátěžových testů nám ukazuje tabulka 6.2, můžeme tak vidět, že vkládané prvky do kolekce při testování budou i zde typu `Integer`, prvky budeme vkládát a přistupovat k nim ve čtyřech možných sekvencích, a to vzestupně, sestupce, procházením *hard* sekvence nebo náhodně.

Velikost kolekce budeme měnit od hodnoty `n` = 100 prvků až po hodnotu 100 000. Vícevláknový přístup nás bude zajímat stejně jako u kolekcí typu `Map`, až pro 24 vláken, které budou dané seznamy sdílet.

Při pohledu na výše uvedené návrhy testů můžeme vidět, že zatížení datových kolekcí bude opravdu značné. Pro každou jednu kolekci typu `Map` se spustí 260 testů pokrývající všechny kombinace pro různé operace a různé sekvence při zvyšování velikosti kolekce (kdy uvažujeme konstantní počet současně přistupujících vláken) a počtu vláken současně přistupujících ke kolekci (kdy budeme naopak počítat s konstantní velikosti kolekce). U kolekcí typu `List` se takových testů provede 208.

Zajímavé budou také jistě výsledky pro různé typy procesorů disponujících s menším nebo vyšším počtem jader, které budou při zvyšujícím se počtu vláken (až do 24) zajišťovat paralelní nebo pseudoparalelní přístup.

Abych dosáhl co nejpřesnějších výsledků, bude třeba každý test co nejvíckrát zopakovat, což nám napoví o tom, jak byla jednotlivá měření výkonu přesná a jaká je směrodatná odchylka.

6 Implementace zátěžových testů datových kolekcí v jazyce Java

Šestá kapitola popisuje samotnou implementaci navržených testů z předešlé kapitoly. Projekt Testování výkonnosti Java kolekcí na vícejádrových systémech byl vyvíjen v jazyce Java ve verzi 6.0 ve vývojovém prostředí NetBeans IDE 7.1.

Jedná se o konzolovou aplikaci, která spouští různé konfigurace zátěžových testů na základě parametrů zadaných na příkazové řádce. Po dokončení testů se naměřené výsledky tisknout na standardní výstup.

6.1 Zpracování parametrů příkazového řádku

Aplikace při spuštění očekává povinné parametry, které zásadně ovlivňují celkový běh programu. Určují, pro jaké rozhraní se budou zátěžové testy spouštět, jaké operace se budou testovat a další konkrétní konfigurace, které vycházejí z návrhu (viz kapitola *5 Návrh vhodných zátěžových testů pro měření výkonnosti datových kolekcí*). Přesnou definici parametrů si můžeme prohlédnout po vytisknutí nápovědy k programu, která obsahuje návod k použití:

```
Usage: JCPerformanceTesting --help
   or: JCPerformanceTesting --interface=list|map --method=method --size=N --seq=sequence --threads=N
Where options include:
       --interface=list|map  test of collections implementing the interface list or map
       --method=method       method of collection that will be tested
                             List[add, get, set, remove];  Map[put, get, key, value, remove]
       --size=N              size of tested collection
       --seq=sequence        use ascending, descending, hard or random sequence
       --threads=N           number of threads accessing the collection
```

V návodu k použití vidíme, že program příjmá dlouhé parametry typu *--parametr=<argument_parametru>*, které jsou hned na začátku hlavní funkce zpracovány pomocí metody `getParams()` a jejich argumenty uloženy do struktury `Settings`. Tato struktura uchovává informace o zadaných parametrech a používá se k vyhodnocení rozhodujících podmínek v dalších částech programu. Pokud uživatel zadá špatné parametry nebo zapomene uvést některý z povinných parametrů, tak program vypíše chybovou hlášku a ukončí se.

6.2 Vytvoření datových kolekcí

Jednotlivé objekty, které reprezentují datové kolekce, jsou ještě zapouzdřeny ve strukturách `MyListCollections` a `MyMapCollections`. Vytvářím zde kolekci typu Map, do které vkládám testované kolekce a která se pak prochází po jednotlivých instancích. Při konstrukci kolekcí používám pro kolekce, které nepodporují vícevláknový přístup, metody `synchronizedMap` a `synchronizedList` vracející zapouzdřené synchronizované kolekce. U kolekcí typu `ConcurrentHashMap` a `CopyOnWriteArrayList` není toto zapouzdření zapotřebí. Kolekce `FastMap` poskytuje možnost při vytvoření použít metodu `shared()`, která nastavuje její status jako sdílená.

6.3 Vytvoření vláken a jejich synchronizace na bariéře

Vlákna v mém projektu jsou reprezentována třídami `MyMapThread` a `MyListThread`, které rozšiřují třídu `Thread` a jsou tak jejími potomky. Samotná třída `Thread` implementuje rozhraní `Runnable`, ačkoli její metoda `run()` v této třídě nemá žádnou funkci. Definice funkce `run()` je uvedena v potomcích.

V hlavní funkci `main()` si vytvářím pole vláken o stejné velikosti jako je počet vláken zadaný parametrem na příkazovém řádku (viz kapitola *6.1 Zpracování parametrů příkazového řádku*). Průchodem pole vláken v cyklu `for` se pomocí metody `start()` vytvoří jednotlivá vlákna. Program, který běží v hlavním vlákně, čeká až dceřiná vlákna dokončí svoji práci pomocí volání funkce `join()`. Po vytvoření jsou vlákna, ještě předtím než se začnou vykonávat, synchronizována na tzv. *bariéře*, což je synchronizační prostředek dostupný z balíčku `java.util.concurrent`. Bariera je sdílená všemi vlákny a v úvodu metody `run()` dochází k volání její metody `await()`. Tím je zajištěno, že všechna vlákna mají při testování stejné výchozí podmínky. Bariéru využívám také pro sesynchronizování vláken před začátkem testování jednotlivých kolekcí během průchodu kolekce typu `Map` obsahující jednotlivé instance testovaných kolekcí (viz kapitola *6.2 Vytvoření datových kolekcí*).

6.3.1 Třída MyListThread

Třída `MyListThread` je potomkem třídy `Thread`. Určuje definici řídící metody `run()` a představuje implementaci navržených zátěžových testů pro kolekce typu `List` (viz kapitola 5.2 Testy kolekcí implementující rozhraní list). Při konstrukci objektu se konstruktoru prostřednictvím parametrů předává struktura `Settings` (viz kapitola *6.1 Zpracování parametrů příkazového řádku*), odkaz na sdílenou bariéru, dále se předává odkaz na testované kolekce a na pole, do kterého vlákna zapisují své naměřené výsledky, a jako poslední parametr je index vytvořeného vlákna.

6.3.2 Třída MyMapThread

Podobně jako předešlá třída je i `MyMapThread` potomkem třídy `Thread` a představuje implementaci navržených zátěžových testů pro kolekce typu `Map` (viz kapitola 5.1 Testy kolekcí implementující rozhraní Map). Instance této třídy se vytváří stejně jako u třídy `MyListThread`.

6.4 Měření výkonnosti základních operací nad datovými kolekcemi

Hlavním účelem mého projektu je měření výkonnosti základních operací, které datové kolekce Javy poskytují. Pro měření délky trvání testované operace jsem si vytvořil třídu `MyTime`, která zaznamenává systémový čas začátku a konce a zkoumanou dobu vrací v jednotkách milisekund. Spuštěním programu se otestují všechny kolekce pro jednu danou operaci.

6.5 Zpracování výsledků měření a jejich výstup z programu

Zpracování výsledků se provádí ve třídě `Resolver`, která pracuje s polem obsahujícím, zvlášť každým vláknem naměřené, hodnoty, ze kterých se vypočítá aritmetický průměr doby trvání testované operace pro jedno vlákno. Tento výpočet má na starost soukromá metoda `getAvrgTestDuration()`. Zpracované výsledky jsou vytištěny na standardní výstup pomocí metody `printResults()` ve tvaru velikost testovaných kolekcí následovaná, čárkami oddělenými, časy, které byly naměřeny pro danou testovanou operaci u každé kolekce. Jedná se o výstup ve formátu CSV (*comma separated value*), který jsem si zvolil z důvodu možnosti snažšího vyhodnocení výsledků pomocí některého z tabulkových editorů, jako jsou například Calc nebo Excel.

6.6 Ošetření stavových chyb programu

Jako každý správně napsaný program, tak i ten můj ošetřuje možné chybové stavy, jako je například špatně zadaný parametr na příkazové řádce, a poskytuje uživateli informace v případě, když takový stav nastane. Stavové kódy programu spolu taky s jejich popisem definuji ve výčtovém typu `enum`. Aktuální stav programu pak vyjadřuje hodnota stavové proměnné `state` ze struktury `Settings` a pokud je při vyhodnocení podmínky v dané chvíli programu hodnota proměnné různá od `EOK` (což znamená vše v pořádku), tak dojde k ukončení programu s chybou, jejíž popis se vypíše na standardní chybový výstup `System.err`.

6.7 Spouštěcí skripty v Perlu

V kapitole *6.1 Zpracování příkazového řádku* jsem popisoval způsob použití mého programu, který přijímá různé parametry. Parametry určují konkrétní konfiguraci zátěžového testu, který se má vykonat. Pro spouštění různých konfigurací zátěžových testů jsem si napsal spouštěcí skripty v jazyce Perl.

Na základě svého návrhu, popsaného v 5. kapitole *Návhr vhodných zátěžových testů pro měření výkonu datový kolekcí*, se spouští dvě sady testů. První sada se zaměřuje na zvyšování velikosti kolekce po násobcích čísla 10 až do její maximální hodnoty 10 000 (z důvodu časové náročnosti jsem upustil od maximální velikosti kolekce 100 000, jak se uvádí v původním návrhu testování). Počet současně přistupujících vláken ke kolekci je při těchto testech konstantní, který je dán počtem jader, kterým procesor na daném počítači disponuje. Ve druhé sadě testů dochází ke zvyšování počtu vláken při konstantní velikosti kolekce 10 000. Testuje se až pro 24 současně přistupujících vláken ke kolekci. Komplexní výsledky testování spolu s informacemi o verzi virtuálního stroje Javy a prostředí, na kterém byly kolekce testované se ukládají do výstupního souboru v adresáři `output`.

Abychom získali co nejpřesnější výsledky je třeba každý test mnohokrát zopakovat a z výsledků vypočítat průměrné hodnoty a odlišnosti v získaných výsledcích pomocí směrodatné odchylky. Druhý spouštěcí skript provadí opakované spuštění prvního skriptu. Počet opakování, kolikrát se mají spustit sady testů, celé číslo `n` zadané jako parametr, který druhý skript příjmá. Po ukončení se vytvoří soubor se záznamem celého testování (*logfile*).

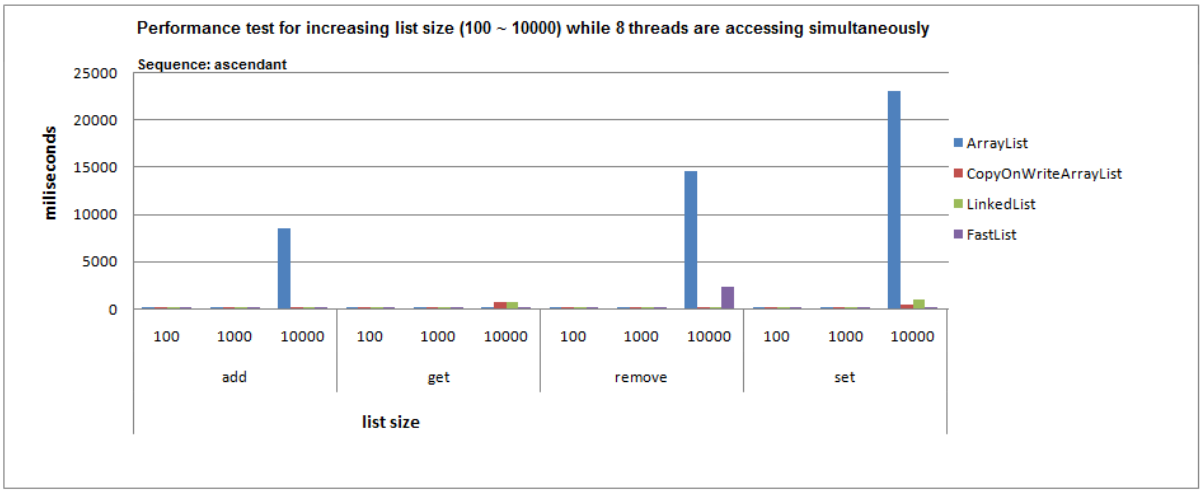
7 Zhodnocení výsledků dosažených při testování výkonnosti Java kolekcí

Předposlední sedmá kapitola se zaměřuje na vyhodnocení naměřených výsledků mé testovací aplikace `JCPerformanceTesting`, jejíž implementace je popsána v předešlé kapitole *Implementace zátěžových testů datových kolekcí v jazyce Java*. Naměřené výsledky byly uloženy do textového souboru v podobě formátu CSV (viz kapitola *6.5 Zpracování výsledků měření a jejich výstup z programu* a kapitola *6.7 Spouštěcí skripty v Perlu*). Vyhodnocení jsem provedl pomocí tabulkového editoru Excelu, který poskytuje pokročilé funkce pro zpracování velkého množství dat. Jednou z takových funkcí je kontingenční tabulka, kterou jsem využil pro práci s agregovanými daty, ze kterých jsem snadno dokázal vypočítat průměr ze všech provedených měření a také směrodatnou odchylku určující chybu měření. Pro dosažení co nejpřesnějších výsledků byl každý jednotlivý test spuštěn minimálně ve 30 opakováních.

Testování se provádělo celkově na šesti různých počítačích, které měly k dispozici vícejádrové procesory s architekturou Intel nebo AMD. Minimální konfigurace byla Intel Core duo (2 jádra) a maximální 4x AMD quadcore (16 jader). Pokud neuvádím jinak, tak všechny výsledky v této kapitole byly naměřeny na počítači s 8 jádrovým procesorem AMD.

7.1 Výkonnost datových kolekcí při zvyšování jejich velikosti

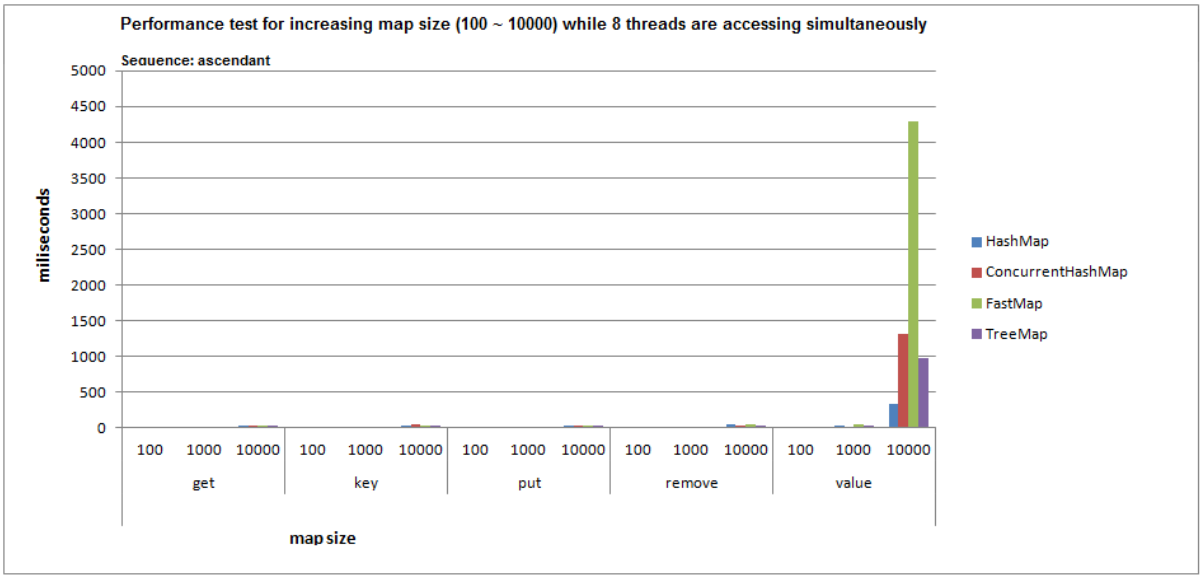
Sada testů zaměřená na měření výkonnosti při zvyšování velikosti datových kolekcí a konstantním počtu současně přistupujících vláken ke kolekci. Jak popisuji v 5. kapitole *Návrh vhodných zátěžových testů pro měření výkonu datovýchkolekcí*, spouští se zde testy pro všechny základní operace a pro různé sekvence.



Graf 7.1: Testování výkonnosti Java kolekcí implementující rozhraní List pro základní operace - add(), get(), set() a remove() - zaměřené na zvyšování velikosti kolekce

Výsledek měření pro rozhraní `List`, kdy jsem pracoval se vzestupně seřazenou sekvencí, je znázorněn v grafu 7.1. Velmi slušných hodnot zde dosahuje implementace z balíku javolution

FastList, který byl nejrychlejší pro operace vkládání (10 000 prvků průměrně za 34,11 milisekund se směrodatnou odchylkou 4,36 ms) a nastavování (10 000 prvků průměrně za 34,73 milisekund se směrodatnou odchylkou 3,29 ms). Druhé nejrychlejší výsledky dosáhnul FastList pro operaci get(), kdy načtení 10 000 prvků trvalo v průměru 35,17 milisekund se směrodatnou odchylkou 5,32 ms. První místo pro operaci get() zde patří kolekci ArrayList, která, při vícevláknovém přístupu ke kolekci, dokázala načíst 10 000 prvků v průměru za 20,4 milisekund se směrodatnou odchylkou 3,45 ms. Naopak pro ostatní operace add(), set() a remove() dosahovala tato, programátory velmi často používaná, kolekce zdaleka nejhorších časů (vkládání 10 000 prvků trvalo průměrně 8 585,38 milisekund se směrodatnou odchylkou 827, 2 ms, operace set() pro 10 000 prvků se provedla v průměru za 23 085,5 milisekund se směrodatnou odchylkou 1 229 ms a operace mazání 10 000 prvků pak v průměru trvalo 14 540,55 milisekund se směrodatnou odchylkou 948,7 ms). Horší výslekdy pro kolekci ArrayList se však daly očekávat. Překvapením rovněž nebylo, že druhý nejhorší výsledek u operace vkládání (10 000 vložených prvků průměrně za 64,72 milisekund se směrodatnou odchylkou 7,46 ms) jsem naměřil u kolekce CopyOnWriteArrayList, která funguje na principu vytváření nové kopie pole a její aktualizaci. Kolekce vykazovaly podobné výkony i pro ostatní sekvence, které se testovaly.



Graf 7.2: Testování výkonnosti Java kolekcí implementující rozhraní Map pro základní operace - put(), get(), containsKey(), containsValue() a remove() - zaměřené na zvyšování velikosti kolekce

Grafická podoba výsledků měření výkonu kolekcí implementující rozhraní Map je znázorněna ve grafu 7.2. Opět se jedná o případ, kdy pracuji se seřazenout sekvencí rostoucí vzestupně od nuly až po hodnotu sto, tisíc nebo deset tisíc.

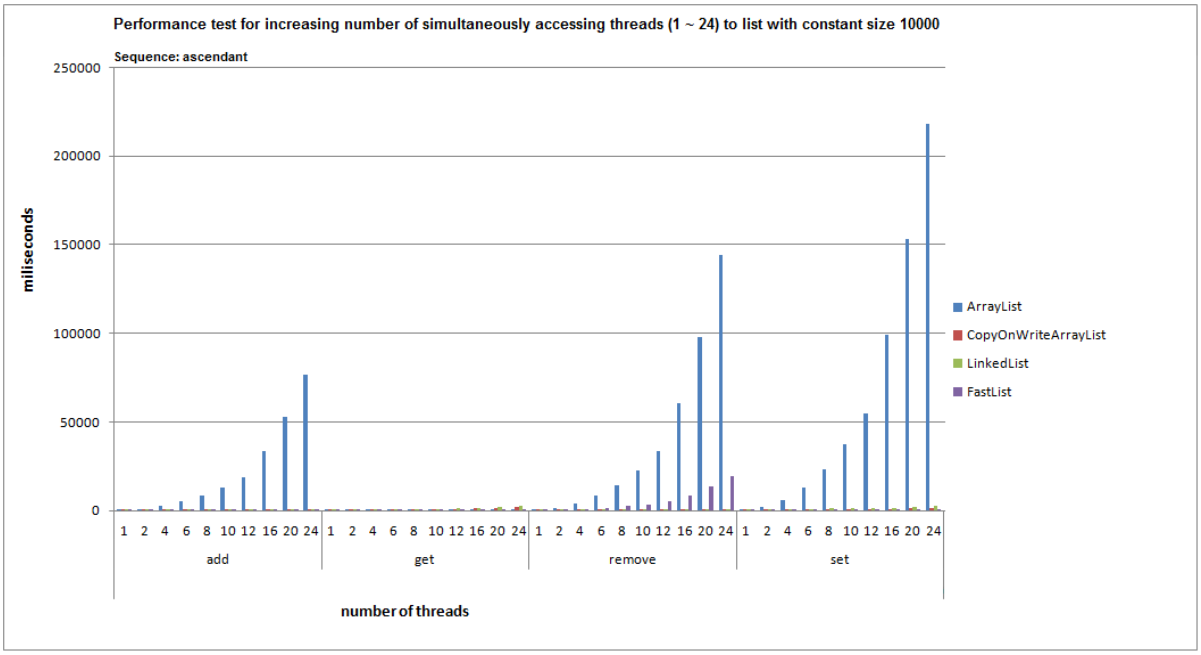
Jedněch z nejlepších výkonů pro operaci vkládání dosahovala z testovaných kolekcí kolekce TreeMap, u které jsem pozoroval průměrný čas 28,37 milisekund pro 10 000 vkládaných dvojic se směrodatnou odchylkou při měření 6 ms. Podobný výsledek zaznamenala i kolekce ConcurrentHashMap, které trvalo vložení stejného počtu dvojic klíč-hodnota průměrně za 28,82 milisekund se směrodatnou odchylkou 3,21 ms. Nejlepší výkon kolekce ConcurrentHashMap, která díky své implementaci implicitní synchronizace (*viz kapitola 4.4.1 ConcurrentHashMap*) podporuje vícevláknový přístup, jsem také očekával. Použití explicitní synchronizace u kolekce TreeMap však předvedlo velice dobré výsledky. Nejhůře z tohoto testu dopadla kolekce FastList z balíku javolution (vložení 10 000 prvků trvalo 37,99 milisekund se směrodatnou odchylkou 4,92 ms). U kolekce se stromovou strukturou jsem při velikosti kolekce 10 000 naměřil nejlepší výsledky i pro operace get() (průměrná doba 21,89 milisekund se směrodatnou odchylkou 6,79 ms), containsKey() (průměrná doba 21,67 milisekund se směrodatnou odchylkou 6,08 ms) a remove() (průměrná doba 33,82 milisekund se směrodatnou odchylkou 5,02 ms). Operace

`containsValue()` se pak provedla nejrychleji u kolekce `HashMap` za průměrnou dobu 339,02 milisekund se směrodatnou odchylkou 5,56 ms, opět při velikosti 10 000 dvojic klíč-hodnota. Stejně jako testování operace `put()`, tak i pro ostatní operace byly nejhorší výsledky naměřeny u kolekce `FastMap`, která při použití příznaku `shared` sice podporuje vícevláknový přístup bez nutnosti další explicitní synchronizace, ale oproti kolekcím `ConcurrentHashMap` nebo synchronizovanému `TreeMap` poskytuje slabší výkon a tedy se moc nehodí k použití jako sdílený prostředek.

7.2 Výkonnost datových kolekcí při zvyšování počtu vláken přistupujících současně ke kolekci

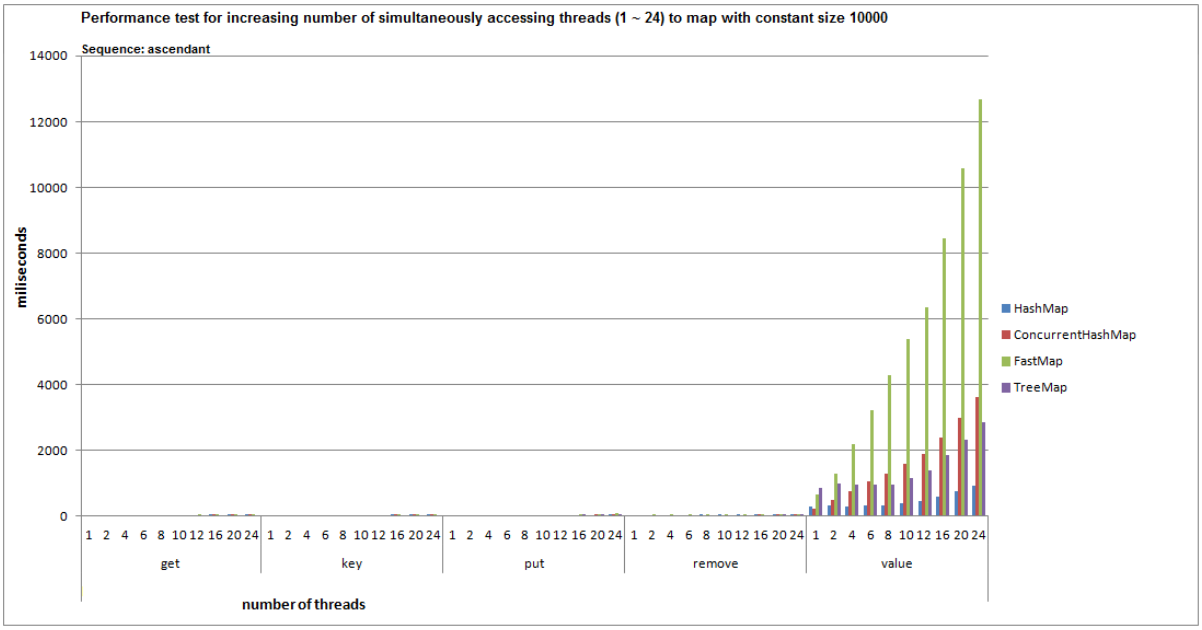
Výkonnost jednotlivých kolekcí při vícevláknovém přístupu je výrazně ovlivněna počtem vláken, které ke kolekci přistupují. V mém testovacím programu spouštím postupně jednotlivé výkonnostní testy pro všechny základní operace a pro čtyři různé sekvence (‚které jsem si určil v 5. kapitole *Návrh vhodných zátěžových testů pro měření výkonu datovýchkolekcí*), přičemž postupně zvyšuji počet generovaných vláken, které ke kolekci přistupují (postupnost zvyšování počtu vláken je rovněž uvedena v 5. kapitole). Pro větší názornost a komplexnost výsledků jsem do testování zahrnul i testy, kdy ke kolekci přistupuje pouze jedno vlákno. Velikost kolekcí je v tomto případě vždy stejná 10 000 prvků/dvojic.

Z naměřených výsledků je patrné, že čím je počet současně přistupujících vláken ke kolekci větší, tím stoupá i časová náročnost prováděné operace, což potvrzuje i můj předpoklad, že složitější režie při synchronizaci jednotlivých vláken bude mít vliv na výkony zkoumaných kolekcí. Testy pro kolekce implementující rohraní `List` jsou jsou vykresleny ve grafu 7.3. Nejrychlejší kolekce `FastList` dosahuje u operace vkládání jedním vláknem průměrného času 3,75 milisekund se směrodatnou odchylkou 0,43 ms. Vkládání pomocí dvou vláken pak trvá všem kolekcím více než



Graf 7.3: Testování výkonnosti Java kolekcí implementující rohraní List pro základní operace - add(), get(), set() a remove() - zaměřené na zvyšování počtu vláken současně přistupujících ke kolekci

čtyřnásobně déle než pomoci vlákna jednoho (průměrný čas pro `FastList` je 17,31 milisekund se směrodatnou odchylkou 2,76 ms, průměrný čas pro `ArrayList` je 1030,53 milisekund se směrodatnou odchylkou 52,8 ms, průměrný čas pro `CopyOnWriteArrayList` je 41,625 milisekund se směrodatnou odchylkou 5,98 ms a průměrný čas pro `LinkedList` je 11,53 milisekund se směrodatnou odchylkou 1,3 ms). Vkládání pomocí čtyř vláken pak trvá skoro osmkrát pomaleji. Pro dvacet čtyři současně přistupujících vláken trvalo `FastListu` vložit 10 000 prvků v průměru 61,71 milisekund se směrodatnou odchylkou 8,1 ms. `CopyOnWriteArrayListu`, který měl nejpomalejší vkládání, to trvalo průměrně 99,88 milisekund se směrodatnou odchylkou 7,45 ms. Při operaci `get()`, kde figuroval na prvním místě `ArrayList` jsem naměřil pro jedno vlákno průměrnou dobu čtení 1,15 milisekund se směrodatnou odchylkou 0.3 ms a pro dvacet čtyři vláken průměrnou dobu 39,77 milisekund se směrodatnou odchylkou 15,16 ms. Při testování operace `set()`, znovu nejrychlejší `FastList` zaznamenal podobné výsledky jako u operace vkládání. O trochu lepší časy, pro jedno vlákno jsem naměřil průměrný čas 3,093 milisekund se směrodatnou odchylkou 0,39 ms. Druhý nejpomalejší byl `FastList` opět při testování operace `remove()`, kdy jsem pro jedno vlákno mazající 10 000 prvků z kolekce naměřil průměrnou hodnotu 42,71 milisekund se směrodatnou odchylkou 0,45 ms a pro dvacet čtyři vláken 19 207,87 milisekund se směrodatnou odchylkou 1 290,62 ms. Nejvýkonnější operaci mazání měla kolekce `LinkedList` s průměrným časem 10,31 milisekund se směrodatnou odchylkou 0,47 ms pro jendo vlákno a 61,02 milisekund se směrodatnou odchylkou 9 ms pro dvacet čtyři vláken.



Graf 7.4: Testování výkonnosti Java kolekcí implementující rozhraní Map pro základní operace - put(), get(), containsKey(), containsValue() a remove() - zaměřené na zvyšování počtu vláken současně přistupujících ke kolekci

Graf 7.4 ukazuje výsledky naměřené při testování kolekcí implementující rozhraní Map. I zde se postupně pro všechny operace testovala výkonnost kolekcí při zvyšujícím se počtu vláken pracující s kolekcí ve stejný okamžik.

Jedna z nejrychlejších kolekcí `ConcurrentHashMap` zde dosahoval průměrných časů při operaci vkládání jedním vláknem 2,53 milisekund se směrodatnou odchylkou 0.5 ms. Tímto se ukázalo, že tato kolekce je opravdu velmi výkonná i při jednovláknovém přístupu. Pro dvacet čtyři vkládájících vláken jsem naměřil průměrnou hodnotu 61,36 milisekund se směrodatnou odchylkou 6,89 ms. Rozdíl mezi jednovláknovým a vícevláknovým přístupem ke kolekcím je docela vysoký. To jsem pozoroval i pro ostatní operace a sekvence.

7.3 Vliv architektury procesoru a jejich výkon při vícevláknovém přístupu k datovým kolekcím

Spouštění testování na různých počítačích s různymi procesory mě inspirovalo k napsání této podkapitoly. Během testování jsem vypozoroval, že na procesorech s různou architekturou Intel nebo AMD trvá běh mé aplikace, která představuje kompletní sadu testů pro obě rozhraní, pro všechny operace a pro všechny sekvence, rozdílnou dobu. Zjistil jsem tedy, že výkon jednotlivých kolekcí může záviset i na typu procesoru, který se na daném počítači nachází. Toto chování si vysvětluji rozdílnou schopnosti plánování procesů na procesoru a rychlosti přepínání kontextu mezi nimi.

Jedno spuštění mé aplikace `JCPerformanceTesting` vykonává testování, které je popsáno v předešlých podkapitolách (viz také kapitola 5. *Návrh vhodných zátěžoých testů pro měření výkonu datovýchkolekcí*). Celkem se během jednoho běhu spustí 468 zátěžových testů spouštěných pomocí spouštěcích skriptů. Porovnání dvou školních počítačů, obsahujících osmijádrové procesory AMD a Intel, popisuje tabulka 7.1:

hostname	JVM Version	OS	CPU Architecture	Available CPUs
ju	java version "1.6.0_18"	Linux	AMD	8
pat	java version "1.6.0_27"	Linux	Intel	8

Měření	čas v sekundách	
1.	14551	2537
2.	14548	2520
3.	14541	2520
4.	14554	2522
5.	14558	2524
Průměr	14550,4	2524,6

Podíl	5,763447675

Tabulka 7.1: Porovnání procesorů AMD a Intel

Pomocí spouštěcího skriptu jsem si změřil také celkovou dobu testování. Z pěti měření jsem si vypočítal aritmetický průměr a zjistil jsem, že doba jednoho běhu testování trvala na počítači se jménem *ju* v průměru 4 hodiny 2 minuty a 30 vteřin se směrodatnou odchylkou 6,42 s. Ty stejné testy, ale spouštěné na počítači se jménem *pat* trvaly průměrně 42 minut a 4 vteřiny se směrodatnou odchylkou 7.1 vtěřin. Z výsleků mi tedy vyplývá, že prováděné testování probíhalo na počítači s procesory Intel v průměru 5,76 x rychleji než na počítačích s architekturou procesorů AMD. Tento výsledek také odpovídá realitě, kterou jsem během testování vypozoroval.

7.4 Možnosti dalšího rozšíření vytvořených výkonnostních testů

Přesto, že testování výkonnosti Java kolekcí na vícejádrových systémech popsané v této práci bylo docela rozsáhlé, dala by se zde navrhnout rozšíření, která by prozkoumaly další chování zkoumaných datových struktur. Práci by bylo možné také rozšířit a další kolekce, které by se přidaly do projektu a jejich výkon by se porovnal s těmi stávajícími, což by rozšířilo možnosti výběru kolekcí pro aplikace s různými účely. Mohlo by se jednat například o kolekce z projektu Guava, který je v Javě vyvíjen společností Google od verze 1.6.

Další rozšíření by mohlo být porovnání různé způsoby explicitní synchronizace kolekcí (já jsem v této práci používal pro účely synchronizace metody `synchronizedList` a `synchronizedMap`). Jako jedna možnost by byla použít explicitní synchronizaci v podobě synchronizačního příkazu použitého při provádění jednotlivých operací. Druhou možností je použít pro synchronizaci zámky z knihovny `java.util.concurrent`. Tyto další možnosti synchronizace by se pak porovnaly s implicitní synchronizací, kterou mají některé kolekce zabudouvanou v rámci své vlastní implementace.

Z hlediska testování výkonů datových kolekcí by se dalo rovněž testovat výkon kolekcí při současném provádění více operací nad danou kolekcí zároveň (například zápis do a čtení z kolekce ve stejný okamžik).

8 Závěr

Během pracování na této bakalářské práci jsem se seznámil s rámcem Java Collection Framework, který slouží pro práci s kolekcemi v programovacím jazyce Java. Nastudoval jsem rozhraní List a Map. Cílem bylo porovnat jednotlivé implementace a popsat jejich rozdíly. Kromě základních datových kolekcí, které poskytuje rámec Java Collection Framework, jsem se seznámil taky z kolekcemi z balíku java.util.concurrent a také se dvěmi kolekcemi s projektu javolution. Při řešení mého projektu jsem náhlédl také do problematiky vláken a jejich synchronizace, což může při špatném návrhu aplikace způsobit vážné problémy.

Hlavním úkolem bylo navrhnout zátěžové testy zkoumající výkon nastudovaných datových kolekcí při vícevláknovém přístupu a tyto testy naimplementovat. Myslím, že stanovené cíle byly splněny a dosažené výsledky jsem zhodnotil v kapitole *Zhodnocení výsledků dosažených při testování výkonnosti Java kolekcí*.

Literatura

[1] DVOŘÁK, Václav. *ARC - Architektury a programování paralelních systémů*. [přednáška]. Brno: FIT VUT, 2. února 2013.

[2] GOETZ, Brian. *Java concurrency in practice*. Upper Saddle River, NJ: Addison-Wesley, c2006. ISBN 0321349601.

[3] Vícevláknové programování: Vlákna. KEOGH, James. *Java bez předchozích znalostí: Průvodce pro samouky*. Brno: CP Books, a.s., 2005, s. 156-173. ISBN 80-251-0839-2.

[4] Kontejnery. PITNER, Tomáš. Java - Začínáme programovat: podrobný průvodce začínajícího uživatele. Praha: Grada Publishing, spol. s.r.o., 2002, s. 125-144. ISBN 80-247-0295-9.

[5] VOJNAR, Tomáš. *Operační systémy - Synchronizace procesů*. [přednáška]. Brno: FIT VUT, 12. dubna 2010.

[6] *Collections Framework Overview* [online]. Oracle and/or its affiliates. [vid. 10.5.2013].
URL
http://docs.oracle.com/javase/6/docs/technotes/guides/collections/overview.html

[7] *Javolution* [online]. Jean-Marie Dautelle. [vid. 10.5.2013].
URL javolution.org

[8] *Package javolution.util* [online]. Javolution. [vid. 10.5.2013].
URL http://javolution.org/target/site/apidocs/javolution/util/package-summary.html

[9] java-collection.jpg. In: *Learn with Harsha* [online]. © 2013 Learn with Harsha. [vid. 10.5.2013].
URL http://learnwithharsha.com/day-5-collections-framework/

[10] *Package java.util.concurrent* [online]. Oracle and/or its affiliates. [vid. 10.5.2013].
URL
http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Seznam příloh

Přiložené CD obsahuje:

- Zdrojové kódy této práce v podobě projektu v IDE NetBeans
- Použitou knihovnu Javolution obsahující kolekce FastList a FastMap
- skripty demontrující použití a činnost
- readme soubor s popisem použití
- text této práce ve formátu PDF